language_model.py

```python
import re
from collections import defaultdict

from arpa import write_arpa_file
from ngram import generate_ngrams


# ===================================================== Text ========================================================= #
def load_text_from_file(file, number_of_sentence_pseudo_words=1):
    with open(file, 'r') as f:
        return preprocess_text(f.read(), number_of_sentence_pseudo_words)


def preprocess_text(text, number_of_sentence_pseudo_words):
    # text preprocessing/adjustment:
    text = text.lower()
    # replace all none alphanumeric characters with spaces
    text = re.sub(r'[^a-zA-Z0-9\s]', ' ', text)
    # every line is a sentence
    lines = text.splitlines()

    for i in range(number_of_sentence_pseudo_words):
        # add "sequence beginning" and "sequence end" pseudo words
        for i in range(len(lines)):
            # add "sequence beginning" and "sequence end" pseudo words
            lines[i] = '<s> ' + lines[i] + ' </s>'

    return lines


# ===================================================== Text ========================================================= #


# =================================================== n-grams ======================================================== #
def calculate_ngrams_stats(ngrams):
    stat_dict = defaultdict(lambda: {'count': 0, 'value': 'not present'})
    for ngram in ngrams:
        # ngram ≔ w_1…w_n = w_1:w_n
        if ngram not in stat_dict:
            # stat_dict[ngram]['count'] ≔ C(w_1…w_n); n-gram count
            stat_dict[ngram] = {'count': 1, 'value': ngram}
        else:
            ngram_stat = stat_dict.get(ngram)
            ngram_stat['count'] += 1
    return stat_dict


def calculate_predecessor_stats(ngrams):
    predecessors = []
    for ngram in ngrams:
        predecessors.append(ngram[:-1])
    predecessor_stats = calculate_ngrams_stats(predecessors)
    return predecessor_stats


# =================================================== n-grams ======================================================== #


def generate_language_model(n, probabilities_function):
    """
    Language model with probabilities calculated from the passed probabilities_function.
    """
    model = []

    for i in range(1, n + 1):
        probabilities = probabilities_function(i)
        model.append(probabilities)
    return model


def generate_mle_language_model(lines, n):
    """
    Language model with maximum likelihood estimation (MLE) ngram probabilities.
    """

    def calculate_mle_ngram_probabilities(n):
        ngrams = generate_ngrams(lines, n)
        ngram_stats = list(calculate_ngrams_stats(ngrams).values())
        if n != 1:
            predecessor_stats = calculate_predecessor_stats(ngrams)

        probabilities = dict()
        for ngram_stat in ngram_stats:
            ngram = ngram_stat["value"]
            if n == 1:
                # for unigrams: divisor := N; count of all words
                divisor = len(ngrams)
            else:
                predecessor = ngram[:-1]
                # for n-grams: divisor := C(w₁…wₙ₋₁); count of n-grams with the same beginning as w₁…wₙ
                # --> n-gram beginning: w₁…wₙ₋₁
                divisor = predecessor_stats.get(predecessor)["count"]

            # for unigrams: ngram_count := cᵢ = C(wᵢ); count of word wᵢ (unigram) in text
            # for n-grams: ngram_count := cᵢ = C(w₁…wₙ); count of n-gram w₁…wₙ
            ngram_count = ngram_stat["count"]

            # Pₘₗₑ(wₙ|w₁…wₙ₋₁) = C(w₁…wₙ) / C(w₁…wₙ₋₁) [JURAFSKY 2008, eqn. 3.12 on p. 5]
            probability = ngram_count / divisor
            probabilities[ngram] = {"value": probability}

        return {"unique_ngram_count": len(ngram_stats), "n": n, "dict": probabilities}

    return generate_language_model(n, calculate_mle_ngram_probabilities)


def count_unique_words(lines):
    return len(calculate_ngrams_stats(generate_ngrams(lines, 1)))


def generate_add_k_language_model(lines, n, k):
    """
    MLE Language model with addtitive/add-k smoothing according to [JURAFSKY 2008, p. 16].
    """
    # unique_word_count := V [JURAFSKY 2008, eqn. 3.21 on p. 14]
    unique_word_count = count_unique_words(lines)

    def calculate_add_k_probabilities(n):
        ngrams = generate_ngrams(lines, n)
        ngram_stats = calculate_ngrams_stats(ngrams).values()  # dic --> list
        if n != 1:
            predecessor_stats = calculate_predecessor_stats(ngrams)

        probabilities = dict()
        for ngram_stat in ngram_stats:
            ngram = ngram_stat["value"]
            if n == 1:
                # word_count := N
                word_count = len(ngrams)
                divisor = word_count
            else:
                predecessor = ngram[:-1]
                divisor = predecessor_stats.get(predecessor)['count']

            # dividend := C(w₁…wₙ) + k
            dividend = ngram_stat['count'] + k
            # for unigrams: divisor := N + k·V
            # for n-grams: divisor := C(w₁…wₙ₋₁) + k·V [JURAFSKY 2008, Eqn. 3.26 on p. 16]
            divisor += k * unique_word_count
            # probability := P₊ₖ(wₙ|w₁…wₙ₋₁) = [C(w₁…wₙ) + k] / [C(w₁…wₙ₋₁) + kV]
            probability = dividend / divisor

            probabilities[ngram] = {"value": probability}

        return {"unique_ngram_count": len(ngram_stats), "n": n, "dict": probabilities}

    return generate_language_model(n, calculate_add_k_probabilities)


def count_possible_extensions(ngrams):
    """
    Count number of possible and unique extensions of a history (w₁…wₙ₋₁) [WAIBEL 2015, p. 39].
    Or in other words: "for Witten-Bell smoothing, we will need to use the number of unique words that follow the
    history" [CHEN 1998, eqn. 15 on p. 13].
    N₁₊(w₁…wₙ₋₁•) ≔ |{wₙ : C(w₁…wₙ₋₁wₙ) > 0}|      ⟵ set cardinality
    """

    history_set = set()
    extension_words = dict()
    for ngram in ngrams:
        history = ngram[:-1]

        if history not in history_set:
            history_set.add(history)
            extension_words[history] = set()

        last_word = ngram[-1]
        extension_words[history].add(last_word)

    history_counts = defaultdict(int)
    for history in extension_words.keys():
        history_counts[history] = len(extension_words[history])

    return history_counts


def generate_witten_bell_language_model(lines, n):
    """
    Language model with Witten-Bell smoothing according to [WAIBEL 2015, p. 39].

    Witten-Bell is a recursive interpolation method.
    recursive interpolation [CHEN 1998, eqn. 12 on p. 11]:
    Pᵢₙₜₑᵣₚ(wₙ|w₁…wₙ₋₁) = λ(w₁…wₙ₋₁)·Pₘₗₑ(wₙ|w₁…wₙ₋₁) + [1 - λ(w₁…wₙ₋₁)] · Pᵢₙₜₑᵣₚ(wₙ|w₂…wₙ₋₁)

    "In particular, the n-th-order smoothed model is defined recursively as a linear interpolation between the n-th-order
     maximum likelihood model and the (n-1)-th-order smoothed model as in equation (12)). [...]

     To motivate Witten-Bell smoothing, we can interpret equation (12) as saying: with probability λ(w₁…wₙ₋₁) we should
     use the higher-order model, and with probability [1 - λ(w₁…wₙ₋₁)] we should use the lower-order model. [...], we
     should take the term [1 - λ(w₁…wₙ₋₁)] to be the probability that a word not observed after the history (w₁…wₙ₋₁) in
     the training data occurs after that history."
    [CHEN 1998, p. 13]
    """

    mle_language_model = generate_mle_language_model(lines, n)
    backoff = dict()

    def calculate_witten_bell_probabilities(n):
        ngrams = generate_ngrams(lines, n)
        predecessor_stats = calculate_predecessor_stats(ngrams)
        mle_probabilities = mle_language_model[n - 1]

        if n != 1:
            possible_extensions_counts = count_possible_extensions(ngrams)

        probabilities = dict()
        # mle_probability ≔ Pₘₗₑ(wₙ|w₁…wₙ₋₁)
        for ngram in mle_probabilities["dict"].keys():
            mle_probability = mle_probabilities["dict"][ngram]
            if n == 1:
                # end recursion at unigram model
                probability = mle_probability['value']
            else:
                history = ngram[:-1]
                history_count = predecessor_stats.get(history)["count"]
                # possible_extensions_count ≔ N₁₊(w₁…wₙ₋₁•): Count of possible (unique) extensions
                # of a history (w₁…wₙ₋₁).
                possible_extensions_count = possible_extensions_counts[history]

                # Witten-Bell interpolation weights wb_lambda ≔ λ(w₁…wₙ₋₁) [CHEN 1998, eqn. 16 on p. 13]:
                #  [1 - λ(w₁…wₙ₋₁)] = N₁₊(w₁…wₙ₋₁•) /  [N₁₊(w₁…wₙ₋₁•) + C(w₁…wₙ₋₁)]
                wb_lambda = -1 * ((possible_extensions_count / (possible_extensions_count + history_count)) - 1)

                backoff_probability = backoff[n - 1][history]
                backoff_probability["backoff-weight"] = (1 - wb_lambda)

                # Pₗ(wₙ|w₁…wₙ₋₁) = λ(w₁…wₙ₋₁)·Pₘₗₑ(wₙ|w₁…wₙ₋₁) + [1 - λ(w₁…wₙ₋₁)] · Pₗ(wₙ|w₂…wₙ₋₁)
                probability = wb_lambda * mle_probability['value'] + (1 - wb_lambda) * backoff_probability["value"]

            probabilities[ngram] = {'value': probability}

        backoff[n] = probabilities
```

```python
        return {"unique_ngram_count": mle_probabilities['unique_ngram_count'], "n": n, "dict": probabilities}

    return generate_language_model(n, calculate_witten_bell_probabilities)


def count_continuations(ngrams):
    """
    "The continuation count of a string · is the number of unique single word contexts for that string ·."
    [JURAFSKY 2008, p. 21]

    N₁₊(•w₂…wₙ) := |{w₁ : C(w₁…w₂wₙ) > 0}| [CHEN 1998, p. 17]
    """

    successor_set = set()
    continuation_words = dict()
    for ngram in ngrams:
        successor = ngram[1:]

        if successor not in successor_set:
            successor_set.add(successor)
            continuation_words[successor] = set()

        first_word = ngram[0]
        continuation_words[successor].add(first_word)

    continuation_counts = defaultdict(int)
    for successor in continuation_words.keys():
        continuation_counts[successor] = len(continuation_words[successor])

    return continuation_counts


def generate_kneser_ney_language_model(lines, n):
    """
    Language model with Kneser-Ney smoothing according to [JURAFSKY 2008, p. 19].
    """

    # memorize highest ngram order for the Kneser-Ney count cₖₙ
    highest_ngram_order = n

    words = generate_ngrams(lines, 1)
    unique_words = list(set(words))

    ngrams_dict = dict()
    continuation_counts_dict = dict()
    extension_counts_dict = dict()
    ngram_stats_dict = dict()

    for i in range(1, n + 1):
        ngrams_dict[i] = generate_ngrams(lines, i)

    for i in range(1, n + 1):
        ngram_stats_dict[i] = calculate_ngrams_stats(ngrams_dict[i])

        if i != 1:
            continuation_counts_dict[i] = count_continuations(ngrams_dict[i])
            extension_counts_dict[i] = count_possible_extensions(ngrams_dict[i])

    def kneser_ney_count(ngram):
        """
        kneser_ney_count := cₖₙ [JURAFSKY 2008, eqn 3.41 on p. 21]
        "the definition of the count cₖₙ(·) depends on whether we are counting the highest-order n-gram being
         interpolated (for example trigram if we are interpolating trigram, bigram, and unigram) or one of the
         lower-order n-grams (bigram or unigram if we are interpolating trigram, bigram, and unigram) [...]

         The continuation count of a string · is the number of unique single word contexts for that string ·."
        """

        print("kneser ney count for: " + str(ngram))
        ngram_order = len(ngram)
        if ngram_order == highest_ngram_order:
            print("ngram count")
            return ngram_stats_dict[ngram_order][ngram]["count"]
        else:
            print("continuation count")
            return continuation_counts_dict[ngram_order + 1][ngram]

    backoff = dict()

    def calculate_kneser_ney_probabilities(n):
        ngrams = ngrams_dict[n]
        unique_ngrams = list(set(ngrams))

        # discount ≔ d [JURAFSKY 2008, p. 20]
        discount = 0.75

        probabilities = dict()

        for ngram in unique_ngrams:
            if n == 1:  # recursion base
                """
                End recursion at zerogram (0th-order) model ⟶ Interpolating unigrams with zerograms.


                "To end the recursion, we can take the smoothed 1st-order model to be the maximum likelihood
                 distribution, or we can take the smoothed 0th-order model to be the uniform distribution [1/V], where
                 the parameter ε is the empty string."
                [JURAFSKY 2008, p. 11]
                """

                unique_word_count = len(unique_words)

                # kn_lambda_epsilon ≔ λ(ε) = [d / Σᵥ(C(v))] · |w':C(w') > 0|
                kn_lambda_epsilon = (discount / len(words)) * unique_word_count

                # [JURAFSKY 2008, eqn. 3.42 p. 21]
                # probability ≔ Pₖₙ(w) = [max(cₖₙ(w) - d, 0) / Σᵥ(cₖₙ(v))] + λ(ε) · 1/V
                probability = (max([kneser_ney_count(ngram) - discount, 0]) / len(words) +
                               kn_lambda_epsilon / unique_word_count)

                # unk_probability ≔ λ(ε) /  V
                unk_probability = kn_lambda_epsilon / unique_word_count
                probabilities[('<unk>',)] = {"value": unk_probability}

            else:
                history = ngram[:-1]

                # possible_extensions_count ≔ |w':C(w₁…wₙ₋₁w') > 0| = N₁₊(w₁…wₙ₋₁•)
                possible_extensions_count = extension_counts_dict[n][history]

                # [JURAFSKY 2008, p. 21]
                # kn_lambda ≔ λ(w₁…wₙ₋₁) = [d / Σᵥ(C(w₁…wₙ₋₁v))] · |w':C(w₁…wₙ₋₁w') > 0|
                kn_lambda = ((discount / sum(ngram_stats_dict[len(ngram)][history + v]["count"] for v in unique_words))
                             * possible_extensions_count)

                backoff_probability = backoff[n - 1][history]
                backoff_probability["backoff-weight"] = kn_lambda

                # [JURAFSKY 2008, eqn. 3.40 p. 21]
                # probability ≔
                #   Pₖₙ(wₙ|w₁…wₙ₋₁) = [max(cₖₙ(w₁…wₙ) - d, 0) / Σᵥ(cₖₙ(w₁…wₙ₋₁v))] + λ(w₁…wₙ₋₁) · Pₖₙ(wₙ|w₂…wₙ₋₁)
                probability = (max([kneser_ney_count(ngram) - discount, 0]) /
                               sum(kneser_ney_count(ngram[:-1] + v) for v in unique_words) +
                               kn_lambda * backoff_probability["value"])

            probabilities[ngram] = {"value": probability}

        backoff[n] = probabilities
        return {"unique_ngram_count": len(unique_ngrams), "n": n, "dict": probabilities}

    return generate_language_model(n, calculate_kneser_ney_probabilities)


lines = load_text_from_file("sample_text/sample.text")
language_model = generate_kneser_ney_language_model(lines, 3)
write_arpa_file(language_model, "kneser_ney.lm")
```